securityandtrust.lu

A Study of Potential Component Leaks in Android Apps

Li Li	University of Luxembourg / SnT / SERVAL, Luxembourg
Kevin Allix	University of Luxembourg / SnT / SERVAL, Luxembourg
Daoyuan Li	University of Luxembourg / SnT / SERVAL, Luxembourg
Alexandre Bartel	TU Darmstadt / EC SPRIDE, Germany
Tegawendé F. Bissyandé	University of Luxembourg / SnT / SERVAL, Luxembourg
Jacques Klein	University of Luxembourg / SnT / SERVAL, Luxembourg

4 June 2015

978-2-87971-141-6

A Study of Potential Component Leaks in Android Apps

Li Li*, Kevin Allix*, Daoyuan Li*, Alexandre Bartel†, Tegawendé F. Bissyandé*, Jacques Klein*
*SnT, University of Luxembourg, firstName.lastName@uni.lu
†EC SPRIDE, Technische Universität Darmstadt, firstName.lastName@ec-spride.de

***Abstract*—We discuss the capability of a new feature set for malware detection based on potential component leaks (PCLs). PCLs are defined as sensitive data-flows that involve Android inter-component communications. We show that PCLs are common in Android apps and that malicious applications indeed manipulate significantly more PCLs than benign apps. Then, we evaluate a machine learning-based approach relying on PCLs. Experimental validation show high performance with 95% precision for identifying malware, demonstrating that PCLs can be used for discriminating malicious apps from benign apps.**

By further investigating the generalization ability of this feature set, we highlight an issue often overlooked in the Android malware detection community: *Qualitative* aspects of training datasets have a strong impact on a malware detector's performance. Furthermore, this impact cannot be overcome by simply increasing the *Quantity* of training material.

I. Introduction

Recent statistics from Google show that 1.5 million Android devices are activated every day [12], running a wide variety of application in many different usage scenarios. The GooglePlay Store[1] alone currently hosts about 1.5 millions apps [5], some of which are known to behave maliciously [3]. The November 2014 Threats Report [30] from McAfee states that the total number of mobile malware samples exceeded five million, growing by 112% in one year. Indeed, mobile devices are a popular target for attackers, and app markets are still abused by malware developers for spreading their malicious apps. Consequently, the security guard of such markets have become an essential challenge for both end users and market maintainers.

Machine learning techniques, by allowing to sift through large sets of apps to detect malicious apps, appear to be promising for large-scale malware detection and eventually to keep malicious apps from entering app markets [2]. State-of-the-art machine learning approaches for Android malware detection mainly differ in the feature sets that are considered for training the classifiers. For example, Canfora et al. [9] rely on system calls and permissions while Gascon et al. [16] use function call graph properties. Other examples of recurrent feature sets include Java code properties, Intent Filter information, strings, etc. Recently, MUDFLOW [8] has proposed to extract behavioral features by taking into account sensitive data flows in Android apps to identify malware. Although most approaches from the literature exhibit high performance results, there are few cases where authors assess the generalization of such results. In particular, we question the extent of dependency between the training data and the yielded classifiers. We advocate, through an investigation into an example of a new feature set, that the assessment of feature sets must dig into the composition of training datasets.

We study the capability of specific sensitive data-flow features to be discriminative in Android malware detection as in MUDFLOW. Contrary to MUDFLOW, for which the source and sink of the data-flow are necessarily within a single component, we consider data-flows that may led to leaks between two components such as data-flow coming from a source and going out of the component without knowing yet if the related data will go to a sink. Indeed, these potential component leaks (PCLs) are meaningful characteristics of malware since researchers have shown that the inter-component communication (ICC) mechanism introduces a lot of vulnerabilities (e.g., Activity Hijacking) [11], [32].

In our previous work, we have developed PCLeaks [25], a tool for detecting potential component leaks involving two components. For the purpose of this study, we have extended PCLeaks to take into account the case where more than two components are involved in the leak (e.g., one component is used as a bridge component between two others).

This paper reports on an empirical investigation into potential component leaks in Android apps. Eventually, we assess the relevance of potential component leaks as features for machine learning-based Android malware detection. For instance, we experimentally check whether the performance achieved with these features can be generalized to different clusters of Android apps.

The contributions of this paper are as follows:

- We present a discussion on the different types of potential component leaks in Android apps.
- We empirically investigate the distribution of potential component leaks in malicious and benign app datasets.
- We further investigate the discriminative power of PCL-based features for machine learning-based malware detection.

This paper is an extension of a short paper published at the International Conference on Software Quality, Reliability & Security [22]. In the previous version, we introduced PCLs into a new feature set for machine learning-based malware detection. Experiments show that PCLs (as features) achieve high performance, demonstrating that PCLs can be used for discriminating malicious apps from benign apps. Now, we

[1] https://play.google.com/store

would like to check whether the high performance achieved is contributed by the whole training feature set or mainly by a subset of the training set. To that end, in this extended version, we perform an investigation on the generalization of the PCLs-based feature set.

The remainder of this paper is organized as follows. Section II provides preliminary materials about Android ICC and PCLs. Then, we describe our research questions and empirical study methodology in Section III. Next, we present the experimental setup in Section IV and our empirical results in Section V. After discussing threats to validity in Section VI and relating to existing work in Section VII, we conclude with directions for future research in Section VIII.

II. Preliminaries

In this section, we provide necessary background information on Android and on the process for identifying potential component leaks.

A. Inter-Component Communication in Android

Android apps are made of components [4], which are implemented as special Java classes. There are four categories of components: 1) *Activity*, which implements user interfaces; 2) *Service*, which implements background tasks; 3) *Content Provider*, which implements Android specific databases; and 4) *Broadcast Receiver*, which implements event notifications. Any pair of components, from the same category or from different categories, can exchange data and invoke each other using the **I**nter-**C**omponent **C**ommunication (ICC) mechanism. An ICC is typically triggered by one of several specific Android methods which take as parameter a special kind of object, called `Intent`. This `Intent` specifies the target component(s), either explicitly, by setting the name of the target components class, or implicitly, by setting the action, the category and the input data. In order to receive implicit Intents, target components need to declare their capabilities through an `Intent Filter` so that the Android system may match them when requested by a given component.

In recent works, Chin et al. [11] and Octeau et al. [32] have shown that the ICC mechanism is an opportunity for exploiting vulnerabilities in Android apps. In this paper, we consider such vulnerabilities by investigating ICC-based information leaks.

B. Potential Component Leaks

In a previous work [24], we have shown that ICCs are used to leak sensitive data across components. Any component can potentially participate in a leak, for instance by retrieving a piece of sensitive information, by sending this information, or simply by playing the role of a bridge between two other components. Thus when performing static analysis of one single component, some of the data-flow paths, leaking data across the boundary of the component can be identified. In this paper, those data-flow paths are called *Potential Component Leaks* (PCLs). A PCL is not *per se* a leak but it might be exploited by other components and eventually contribute to a leak of private data.

Fig. 1 illustrates four different scenarios of data leakages, three of which represent PCLs. In this figure, (A) represents the "traditional" intra-component privacy leaks, which have been well studied in the literature [7], [17], [19]. In the present study, we do not consider such leaks since they are fully contained in one single component—i.e. a piece of data is both obtained and leaked inside one component—and hence do not involve any Inter-Component Communication. In (B), the leaked data is exfiltrated by the component, while in (C) the data is obtained from the component. Finally, in (D), the leaked data travels through the component which is merely used as a bridge. From these schematic examples, we see that a component is involved in a PCL either by providing an *entry-point* or by providing an *exit-point* for leaking the data.

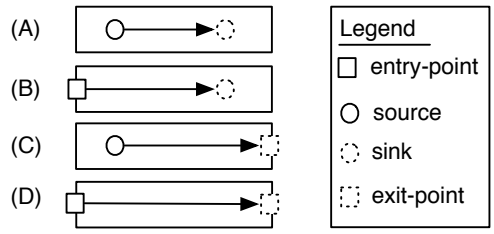


Fig. 1: Examples of leak schemes including "traditional" intra-component leaks (A) and PCLs ((B), (C) and (D)).

1) Characterization of Entry and Exit points: Unlike traditional Java apps, which come with a single entry point (*main* method), an Android app includes several components, each of which may contain several entry-points for launching the app. Because components can use different methods to call out other components, each component may contain multiple exit-points.

We now detail the criteria for identifying such *entry-points* and *exit-points*.

Entry-points: Entry-points are methods where data can be transferred into a component, through parameters, between components. In our study, we consider the following methods:

- Any method such as `getStringExtra()` that obtains data from Intents (or Bundles).
- Any lifecycle method that takes an Intent as a parameter[2].
- Any method that obtains data from `ContentValues` [3].
- Any method of `ContentResolver` such as `query()` that acquires data from other components (or apps).

Exit-points: An exit-point is a method call through which data can be transferred outside a component. For example, the `startActivity()` method can be used to trigger data exchange when one component launches another. We consider all such ICC methods as exit-points. We also take into account such methods of `ContentResolver` such as `insert()` that are capable to transfer data to `ContentProvider`s.

[2] It is not necessary for *entry-points* to get data from Intents since Intents can be directly leaked through components, e.g., type (D) in Fig. 1

[3] Like `Intents`, `ContentValues` are used to exchange data between components. However, they are used to transfer data to `ContentProviders` while Intents are used for the other three components.

2) PCL types: In this section, we introduce the three types of PCLs that are investigated. An example of PCL is shown in Fig. 2, where a sensitive data (device id) is collected in a first component (1) and leaked through an ICC method to another component (2) which simply forwards it to a third component (3) where it is eventually leaked outside the device through SMS. To detect such leaks, PCLeaks performs static taint analysis on each app and tracks the sensitive data across components from its source to a sink.

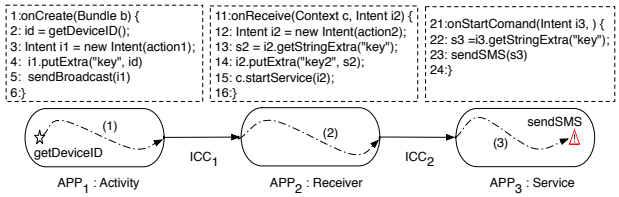


Fig. 2: An example of PCLs.

Potential Active Component Leak (PACL). We define a PACL as a taint flow path starting from a *source* (defined as calls into resource methods returning non-constant values into the application code [35]) and ending with an *exit-point*. Such PCLs are referred to as "active", as the involved component is actively leaking sensitive data that it collected itself to other components (cf. (1) in Fig. 2).

Potential Bridge Component Leak (PBCL). We define a PBCL as a taint flow path starting from an *entry-point* and ending with an *exit-point*. Such PCLs are referred to as "bridge", as the involved component is transferring sensitive data collected by a different component to another component (cf. (2) in Fig. 2).

Potential Passive Component Leak (PPCL). We define a PPCL as a taint flow path starting from an *entry-point* and ending with a *sink* (defined as calls into resource methods accepting at least one non-constant data value from the application code as parameter, if and only if a new value is written or an existing one is overwritten on the shared resource (e.g., GSM network) [35]). Such PCLs are referred to as "passive", as the involved component is passively leaking sensitive data collected by other components (cf. (3) in Fig. 2).

Note that the Android system provides two types of mechanism to protect components of being misused by other components: the *export* attribute and permissions. i) The *export* attribute is used to express the fact that other components can "access" the exported one. Thus, only exported components can potentially leak private data (PPCL and PBCL). ii) Permissions can be used at component level. When a component is protected by a permission, the apps that want to access this component must have first requested, and be granted this permission. In our detection of PCLs, we take into account these two types of mechanism, for instance by checking whether the export attribute is used or not.

III. Research Questions & Measurements

In this section, we provide details on the metrics and assessment methods that we use in this study as well as the research questions that we investigate.

A. Research Questions

In this study, we address the following research questions:

- RQ1: Are PCLs common in Android apps?
- RQ2: Is there a significant difference in the presence of PCLs between malicious and benign apps? If so, is this difference similar for all PCL types?
- RQ3: Can PCLs be used as features for machine learning-based malware detection?
- RQ4: Can we generalize our findings on potential malicious use of PCLs?

B. Metrics and Assessment methods

We now briefly present some methodologies we use in this empirical study. In particular, we provide details on how to interpret the data (e.g., boxplot and MWW test results).

Boxplot. Boxplots are a convenient way to visually illustrate groups of numerical data. As shown in Fig. 3, a boxplot is made up of 5 main horizontal lines. From left to right, they are the least value (MINIMUM), the line where 25% of data points are below (LOWER QUARTILE), the middle of the dataset (MEDIAN), the line where 25% of data points are above (UPPER QUARTILE), and the greatest value (MAXIMUM). Such data points that out of the range of the first and fifth line are outliers. In this paper, we use the R statistical analysis tool to draw boxplots. In order to highlight the difference between median values, we have disabled outliers and for some boxplots we have cut off their upper whiskers.

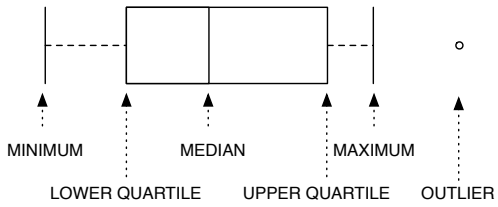


Fig. 3: An example of boxplot.

Mann-Whitney-Wilcoxon (MWW) test. The MWW test is a non-parametric statistical hypothesis test that assesses the statistical significance of the difference between the distributions in two datasets [29]. Given two independent samples x and y, of size n_1 and n_2 respectively, the formula for computing the Mann-Whitney U for x is: $U = n_1 n_2 + \frac{n_1(n_2+1)}{2} - T$. [33]. Once the U value is computed it is used to determine the p-value. Given a significance level $\alpha = 0.001$, if p-value $< \alpha$, then the null hypothesis is rejected by the test, indicating that the two datasets have different distributions at the significance level of $\alpha = 0.001$. In other words, there is one chance in a thousand that this is due to a coincidence.

10-fold cross validation. Cross validation is a validation technique, which is usually used to evaluate how the results

of a statistical analysis will generalize to an independent data set. Mainly, it is used to estimate how accurately a predictive model will perform in practice [21]. The 10-fold cross validation consists in randomly partitioning the given data samples into 10 sub-samples of equal size. Of the 10 sub-samples, 9 of them are used to train a classification model, while the left one is used as the validation data for testing the model. The cross-validation process is then repeated 10 times in which each sub-sample is used exactly once as the validation data. The 10 results from the folds can then be averaged to produce a single estimation.

IV. Experimental Setup

In this section we detail the settings used in PCLeaks to yield PCLs. We also present the dataset for the experiments as well as the construction of the feature vectors for machine learning experiments.

A. PCLeaks Settings

In this study, all PCLs are detected by a new version of PCLeaks [25] which has been extended to detect PBCLs as well. PCLeaks uses a static taint analysis approach to track data paths from sources to sinks. We discuss in this section how such sources and sinks are determined in our work.

Sensitive sources and sinks. The key idea behinds static taint analysis is to identify a path that starts from a sensitive *source* and ends with a sensitive *sink*. In this study, the sensitive *source* and *sink* we use are extracted by SUSI [35], which automatically classifies all methods in the whole Android API as source, sink or neither. In Android 4.2, SUSI yields 18,076 *source* methods and 8,314 *sink* methods. Theoretically, there are 150,283,864 (18,076 * 8,314) different taint paths (the *source* to *sink* pairs) PCLeaks can report. Instead of identifying PCLs by a pair of methods, we group methods with similar functionality into categories (e.g., group methods *Log.e()* and *Log.v()* into category `LOG`) and use this category in lieu of the fully-qualified method name. This categorization allows to vastly reduce the number of different identifying pairs down to a more manageable value. We use the categories provided by SUSI for our study. Note that we have ignored methods that are classified as `NO_CATEGORY` by SUSI except for methods related to *shared preferences* since they are well used in Android apps, for which we create a new category (`SHARED_PREFERENCES`). Besides, we create four new categories (one for each component type) to further break down the behavior of potential component leaks.

Analysis Settings. As mentioned in [25], PCLeaks leverages the static taint analysis tool FlowDroid to identify data flows in Android apps. Because FlowDroid analyzes a whole Android app and aims to provide highly precise results, it usually takes a lot of time and resources to analyze an app. In favor of a faster analysis, we use the same FlowDroid settings as MUDFLOW [8] chooses (*Explicit flow only*, *Disable flow-sensitive alias search*, *Maximum access of path length of 3*, *No-Layout mode* and *No static fields*[4]) , which sacrifices some amount of precision for speed and memory. As a result, the detected potential component leaks may have false positives as well as false negatives. However, our goal in this paper is not to prove the presence or absence of flows but to study the distribution difference of potential component leaks between malware and goodware.

Advertisement Libraries. Most Android apps are free, they usually use advertisement to get profit, which are delivered through specific *advertisement libraries*. These libraries access sensitive data such as the unique device id to deliver personalized advertisements. However, the potential component leaks (flows) introduced by advertisement libraries are separate from the actual app code. As shown in MUDFLOW [8], advertisement libraries are frequently used and their flows (PCLs) thus become "normal", diluting the impact of actual app flows. Therefore, we follow MUDFLOW's assumption that advertisement libraries are trustworthy and ignore all the PCLs taking place in advertisement libraries, allowing our study to focus on the actual app PCLs. We use the same list of libraries MUDFLOW uses to exclude PCLs.

B. Datasets

For the purpose of our experiments, we collected a dataset of Android apps from Android markets including the official GooglePlay store. For each app, we also retrieved analysis results of anti-virus products hosted by VirusTotal[5].

Then, based on the results of VirusTotal, we build two disjoint sets: One set, noted M (for ***M***alware), containing only malicious apps, and G (for ***G***oodware) containing only benign apps. Each dataset contains 5,000 apps which we evaluate with PCLeaks.

All our experiments are performed on the UL HPC platform [36]. For each Android app, we allocate one core for PCLeaks to analyze it. The Java heap is set to 8 gigabytes and the time out is set to 12 hours. Recall that we start with two data sets containing 5,000 apps each for this study. Because some of them fail (e.g., exception or time out) or do not contain any PCLs, the result of the PCL extraction process contain 2,822 goodware and 3,785 malware, each containing at least one PCL.

C. Feature Set

One goal of this study is to assess if PCLs can be used as features for machine learning-based malware detection to suggest potential malicious apps. Machine learning algorithms cannot directly work on Android apps. Each app must be represented by a vector of properties, called a feature vector in the context of machine learning. In this study, our feature vectors are built with the results (PCLs) of PCLeaks, after analyzing all the apps in our dataset. Let L be the feature

[4]Explanations for these FlowDroid settings can be found at https://github.com/secure-software-engineering/soot-infoflow-android/wiki

[5]we consider an app is malicious if at least 20 different anti-virus products detect it as such. An app is considered benign, or Goodware only if it is not detected by any anti-virus product.

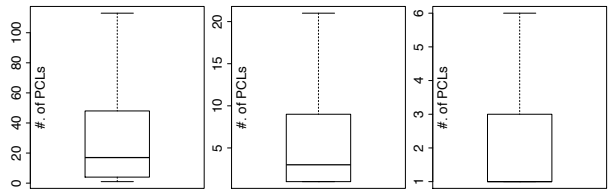
vectors we build, given an app a, for each PCL $l_i \in L$, we value it as either 0 for the case that a does not contain l_i or the actual number of l_i reported by PCLeaks. Recall that we use categorizations instead of methods to describe the detected taint flows (PCLs). Thus, our feature set is made of category pairs. Taking the code in Fig. 2 as an example, we are able to collect the following features:
$UNIQUE\_IDENTIFIER \rightarrow RECEIVER\ (APP_1)$
$RECEIVER \rightarrow SERVICE\ (APP_2)$
$SERVICE \rightarrow SMS\_MMS\ (APP_3)$

V. Empirical Results

We report in this section the results of our investigations to answer the different research questions outlined in Section III.

A. Occurrence of PCLs in Android Apps

Fig. 5a plots the distribution of the number of PCLs per app from our dataset. The median value indicates that half of the apps contain at least 20 PCLs. Excluding outliers, which are automatically identified by the R statistics tool, the number of PCLs per app ranges from 0 to about 100. Because the various apps in our dataset are not equivalent in terms of code size and in terms of components, we further investigate the distribution of PCLs by normalizing the result in those two dimensions. Fig. 5b depicts the distribution of PCLs per 100 kilobyte of bytecode. The median number of PCLs per 100kB is around 3, while the maximum is slightly above 20. Finally, Fig. 5c presents the number of PCLs per component in the apps of the datasets. Components have a median value of 1 PCL, with a maximum of 6 PCLs per component.



(a) Per app (b) Per 100kB (c) Per component

Fig. 5: Distribution of PCLs in Android apps: un-normalized (per app) and normalized densities (per 100kB bytecode and per component)

RQ1:*Although PCLs are common in our datasets, their distribution is uneven across apps and across components.*

B. Distribution of PCLs between malicious and benign apps

Following the findings on the occurrence of PCLs in Android apps in general, we further investigate whether the distribution of PCLs varies between malicious and benign apps. We therefore separately show in Fig. 4a the boxplots representing the number of PCLs for the malware and goodware datasets. The median values indicate that in general half of malware apps contain each more than 22 PCLs while this median value amounts to 8 for in goodware apps. To assess the significance of this difference we perform an MWW test which was successful (with p-value < 0.001).

We then explored whether this difference is similar for all types of PCLs considered in this study. The results show that PACLs, depicted in Figure 4b, are the most unequally distributed between malware and goodware. The median value for PACLs is 15 for malicious apps and only 1 for benign apps. The differences, according to MWW test, although statistically significant, are less important for PPCLs (Figure 4c, median values 4 for malware and 2 for goodware), and PBCLs (Figure 4d, median values 2 for malware and 0 for goodware).

We also explored the categories of leaks identified by PCLeaks and their distribution among the apps. In total, from all the dataset apps, PCLeaks identified 501,320 PCLs. These instances of PCLs can be categorized based on the types of source and sinks involved as well as the type of component involved. Thus, we have identified 74 distinct PCL categories. The top 20 PCL categories, i.e., those with the largest numbers of instances, are depicted in Fig. 6. From this figure, we note that in almost all categories, malicious apps contain more PCLs than benign apps. However, Activity-related PPCLs (e.g., $Activity \rightarrow Shared\_preferences$ and $Activity \rightarrow Log$) are more present in the goodware dataset than in the malware dataset. We note that this kind of potential leaks are user-aware leaks since the source starts from the user interface (i.e., Activity) and the leaked data is actually written to disk.

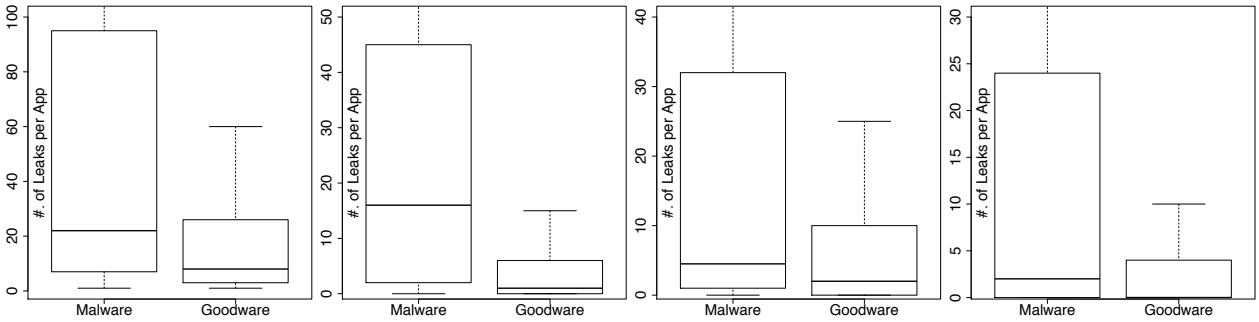
RQ2:*Malicious apps contain significantly more PCLs than benign apps. This difference is most important in the case of Potential Active Component Leaks, i.e., where components actively forward data that they collect outside to other components*

C. Malware identification

Empirical findings from previous section on the presence of PCLs in malware and goodware datasets suggest that PCLs can be used to discriminate malicious apps from benign apps. In this section, we investigate this possibility by implementing and assessing a machine learning-based malware detection approach leveraging PCLs as classification features.

We perform extensive experiments, tuning different machine learning approach parameters, to gather insights for the practical use of PCLs as features. In particular we evaluate the performance of the features in combination with different machine learning classification algorithms. We also consider the impact of class imbalance in the dataset by varying the ratio between malware and goodware in the validation experiments.

Effect of Classification Algorithm. Fig. 7 plots the ROC graphs for the performance of the malware detector with different classification algorithms. All five algorithms yield an Area Under Curve (AUC) above 0.8, indicating good performance. The *RandomForest* algorithm achieves the best performance, although the overall performance of all algorithms are similar.



(a) ALL (b) PACL (c) PPCL (d) PBCL

Fig. 4: PCL distribution across Malware and Goodware datasets

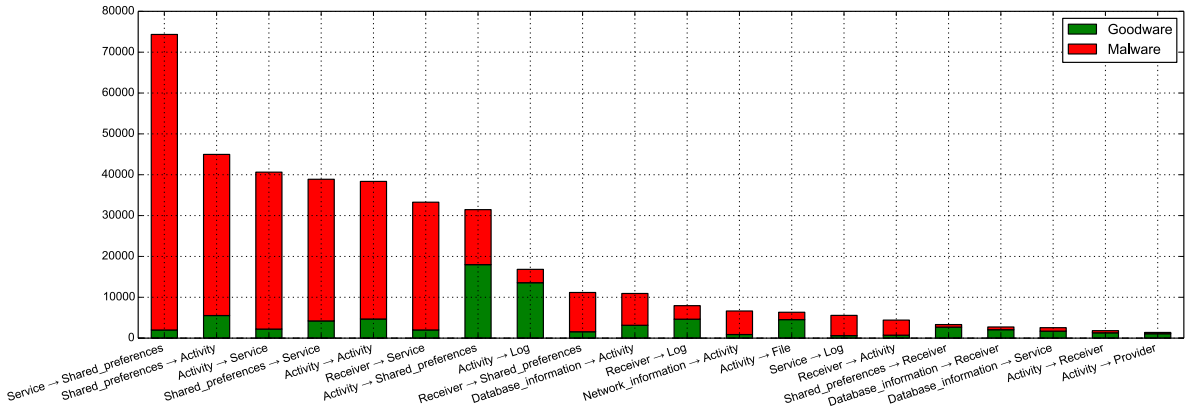


Fig. 6: The top 20 PCLs we detect in our experiments (same number of apps for each set).

This result suggests that the PCL-based feature set is not tailored for a specific algorithm.

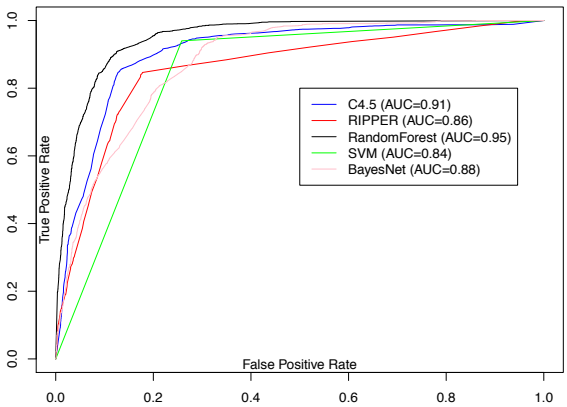


Fig. 7: ROC curves of different algorithms.

Malware/Goodware Ratio. We investigate in detail how class imbalance in the constructed dataset threatens the performance of PCL-based malware classification. To this end, we customize three datasets, composed of 2,400, 3,600 and 4,800 apps with a malware/goodware ratio of 1, 2 and 3 respectively. Fig. 8 depicts the distribution of F-measure for these different malware/goodware ratios. We note that the performance decreases with the ratio of malware in the set. Such a finding was already shown in Allix *et al.*'s large scale empirical study with a different feature set [2].

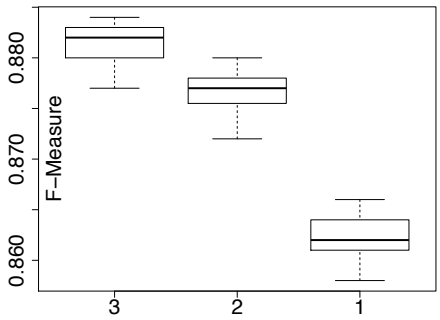


Fig. 8: Distribution of F-Measure for different Malware/Goodware ratios.

RQ3:*PCLs constitute good features for discriminating malicious apps from benign apps in a Machine learning-based malware detection scheme.*

D. Generalization

Generalization is an important criterion for assessing a machine-learning approach, whose validity could be threatened by various steps of the process. For example, features considered may be uncommon in the datasets, or the datasets considered may not be representative of the universe of the relevant artifacts. Unfortunately, the generalization question is often overlooked. In this section we evaluate to what extent PCL-based features can be generalized for detecting malware in the wild (i.e., beyond the scope of the composition of our current experiment datasets) in order to draw insights for specifying the context in which PCLs can be leveraged to construct a feature set for malware detection.

In particular, we investigate whether the performance yielded by the PCL-based classifiers is contributed by the whole training dataset or only by a specific subset. In other words, we explore the possible relationship of PCL-based features with a subset of apps. Thus, we propose to cluster our datasets into different subsets in which apps share similar characteristics. To that end, we consider application permissions as a clustering criterion. Since PCLs are inherently related to sensitive data which is usually protected by a permission, our clustering scheme may cluster apps[6] into subsets that exhibit distinct patterns of PCL usage.

Fig. 9 presents an overview of the experimental process that we have setup in this study.

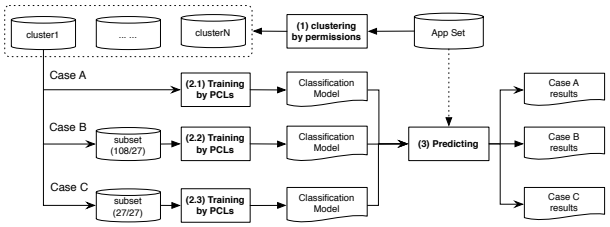


Fig. 9: Experimental setup for assessing generalizability. For cases B and C, the training dataset for each cluster's classifier is randomly selected from the cluster dataset and the experiments are repeated 10 times in order to reduce result variability.

- In step (1), we apply a clustering algorithm on our dataset of Android apps, using the list of permissions requested by each app as a criterion. To that end, we consider the complete list of all permissions appearing in the manifest file of any app in our dataset. Let v be a vector representing the list of permission of a given app a and P the set of all known permissions. For each permission $p_i \in P$, if a has declared p_i in its manifest, then $v[i] =$ "YES", otherwise $v[i] =$ "NO". In our experiments, we found 250 distinct permissions declared in various apps of our dataset. Thus, for each app we build a vector of size 250 indicating the use of certain permissions. We perform different clustering scenarios by varying the number of clusters that the algorithm should output.
- In step (2), we consider the subset of apps in each of the clusters obtained in step (1) to build a classification model, i.e., a classifier. We have identified three cases on how the training dataset can be sampled within the cluster:
  - A. Each classifier is built using all apps within the relevant cluster. In this case, classifiers are trained on sets with characteristics potentially very different, because clusters themselves have neither the same size, nor the same malware/goodware ratio.
  - B. Each classifier is built using training data obtained by sampling apps in the relevant cluster to guarantee the same malware/goodware ratio for all classifiers.
  - C. Each classifier is built using training data obtained by sampling apps of a given cluster in a way such that the ratio between goodware is balanced and the samples sizes are identical across clusters.
- In step (3), we evaluate the classifiers built in step (2) by applying each for detecting malware on the whole app set.

For this study, we use *simpleKMeans* as a clustering algorithm in step (1). Figure 10 illustrates the distribution of potential component leaks for different clusters obtained in step (1). In this figure, we also differentiate between malware and goodware across all 18 clusters. The number of leaks is normalized to take into account the variety of cluster sizes: we divided the number of leaks by the number of apps in each cluster. From this histogram we see that the average number of PCLs per app can be wildly different between clusters, i.e. the patterns of PCL usage is not homogeneous across different groups of apps.

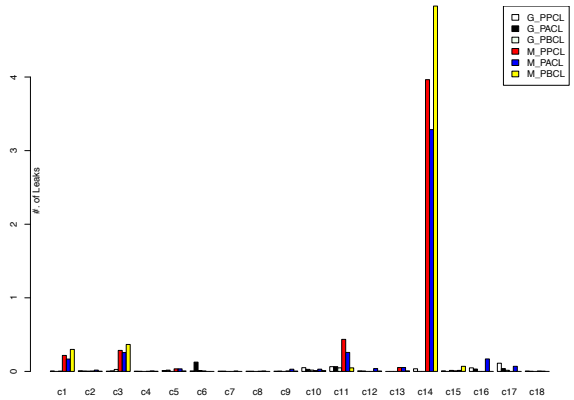


Fig. 10: Distribution of potential component leaks for different clusters.

Experimental results of the machine learning detection tests performed in step (3) are shown in Table I. We focus on precision as a performance criterion for comparing classifiers. For insights on recall, we provide the number of detected malware using each classifier. We also display the performance

[6]For the clustering process, we restrict to such apps that contain at least one PCL.

TABLE I: Assessment results of permission-based clustering. We performed four clustering experiments, varying the number of clusters from 2 to 5.

clusters	Case A			Case B			Case C		
	apps (M/G)	precision	detected malware	apps (M/G)	precision	detected malware	apps (M/G)	precision	detected malware
2	1442 / 2376	0.96	2858	108 / 27	0.67	2743	27 / 27	0.78	2743
	2343 / 446	0.74	3637	108 / 27	0.67	2555	27 / 27	0.81	2555
Baseline	3785 / 2822	0.95	3549	216 / 54	0.7	3515	54 / 54	0.82	2909
3	1116 / 2251	0.97	2780	108 / 27	0.65	2769	27 / 27	0.76	2769
	1090 / 445	0.79	3311	108 / 27	0.67	2660	27 / 27	0.81	2660
	1579 / 126	0.63	3697	108 / 27	0.62	2099	27 / 27	0.84	2099
Baseline	3785 / 2822	0.95	3549	324 / 81	0.7	3553	81 / 81	0.82	2996
4	492 / 37	0.6	3551	108 / 27	0.66	1671	27 / 27	0.81	1671
	1422 / 546	0.79	3564	108 / 27	0.66	2621	27 / 27	0.77	2621
	915 / 67	0.64	3569	108 / 27	0.67	2084	27 / 27	0.8	2084
	956 / 2172	0.97	2733	108 / 27	0.66	2730	27 / 27	0.77	2730
Baseline	3785 / 2822	0.95	3549	432 / 108	0.71	3574	108 / 108	0.84	2862
5	477 / 27	0.59	3534	108 / 27	0.69	1611	27 / 27	0.8	1611
	1348 / 495	0.79	3524	108 / 27	0.67	2764	27 / 27	0.77	2764
	902 / 46	0.63	3639	108 / 27	0.71	2181	27 / 27	0.83	2181
	886 / 2095	0.96	2699	108 / 27	0.65	2843	27 / 27	0.8	2843
	172 / 159	0.7	2481	108 / 27	0.6	1757	27 / 27	0.54	1757
Baseline	3785 / 2822	0.95	3549	540 / 135	0.7	3636	135 / 135	0.81	2917

of a baseline experiment which consists in training on the combination of all sample apps considered for each cluster and testing on the whole dataset of apps. The experiments are repeated 10 times when sampling of training dataset is performed within a cluster.

Case A. The different clustering experiments reveal many insights. In each of the experiments, at least one cluster yields a classifier that provides not only a higher precision than the baseline, but also a significantly higher precision than classifiers built with other clusters.

These performance discrepancies suggest that different clusters provide training material with different quality. However, since the clusters are of different sizes, the differences may simply be related to this aspect, instead of the quality of data.

Case B. With classifiers built with same-size training datasets but drawn from different clusters, we note that different numbers of malicious apps are detected depending on the cluster. In the experiments where the dataset is split into five (5) clusters, we note that one specific cluster provides a 60% precision, which is 10 points lower than the baseline, and recalls roughly 1,000 less malicious apps than some of its counterpart clusters.

Case C. Finally, by considering balanced samples within clusters, we build classifiers whose performance results differ from one cluster to another. E.g., for the experiments where the dataset is split into 5 clusters, the worst classifier yields a precision which is 30 points below the precision of the best classifier and of the baseline. These results confirm the finding that all clusters do not equally contribute to the detection ability of our feature set. Some clusters are constituted with training data that yield a low-precision classifier, indicating that those clusters contain goodware and malware that either a) cannot be discriminated with this feature set or b) are not representative, with regards to this feature set, to the whole universe of apps. In either case, the performance differences amongst clusters suggest that the performance of a malware detector is highly linked to the *quality* of its training set.

Case A vs Case B vs Case C. The results in Table I also provide insights on how the performance of a clusters relates to the quantity of training datasets. Indeed, we note that for Case A and Case B, although the baseline performances are obtained with training data of various sizes they are similar within each Case. In case B the precision of the baseline experiment is 70% for 3 experiments and 71% for the fourth clustering experiment. In Case C, it is between 81% and 84%.

We further note that in Case A, with the whole datasets, i.e. largest and most diversified data, the baseline performance is at his highest point with a precision of 95% and detecting 93% (3549 / 3785) of malware. Case B however, despite its larger training datasets than in Case C, shows worse performance (70% vs ∼80%).

RQ4:*Compared to Baseline, Performance is higher when some parts of the dataset are not used in training. i. e. giving more training data does not always result in better performance.*
Some parts of the dataset are Noise*—with regards to this feature set—meaning that there exist groups of apps this feature set performs poorly on.*

VI. Threats to Validity

We now describe some threats to validity that we have identified in the course of this study.

A. Internal Validity

The size of training sets and the parameters we use (e.g., malware/goodware ratio) take different values that appear to be unjustified since, as shown in [2], no survey has determined the appropriate values for malware detection. However, our results show the same trends of that shown in [2]. For example, we conclude the same trend: the performance of the machine learning-based malware detector decreases when there are fewer malware than goodware in the training data set.

B. External Validity

The main threat to validity in this study is external validity, we now introduce them in this section.

Datasets representativity. The size of the dataset used in the present study is very small compared to the many millions of Android applications in existence. Hence, it could be argued that our dataset has specific characteristics, and that our experiments would yield different results on other datasets. To reduce this risk, our dataset was randomly drawn from a larger dataset whose size is two orders of magnitude bigger.

Furthermore, we show in this paper that sets of apps with specific traits do indeed yield different results. While having a representative dataset is of the utmost importance when claiming experimental results would replicate across various other datasets (i.e., that all sets of apps would be handled as successfully), it is not necessary in our case where we show that not all sets of apps are handled as successfully by a malware detection approach.

Deficiencies of Static Analysis. Like MUDFLOW, our feature set based on potential component leaks is also generated by statically analyzing Android apps. Since we use the same settings as MUDFLOW use (for FlowDroid), our results may contain flows that are unfeasible, as well as miss flows that are feasible. Because one goal of this study is to determine whether PCLs are good features for malware detection, and not to prove the presence or absence of flows, we chose to trade a small amount of precision in favor of a significant speed gain.

VII. Related Work

In this paper, we have studied the distribution of potential component leaks, and used this information to detect malicious apps. This work is related to many existing techniques that leverage static taint analysis to detect privacy leaks, to detect malware or to perform empirical study on Android apps.

Information flow analysis. Information flow analysis has been well studied in Android community to detect vulnerabilities of Android apps. For instance, FlowDroid [7] performs "context-, flow-, field-, object-sensitive and lifecycle-aware static taint analysis for Android apps" to detect intra-component sensitive data flows. Several other works have been presented to detect inter-component information flows [18], [20], [24], [37] and to support inter-app information flow analysis [23]. For example, IccTA [24] leverages FlowDroid to perform inter-component static taint analysis through instrumenting Android apps, reducing an inter-component problem to an intra-component problems. Other techniques dynamically analyzes information flows of Android apps. For example, TaintDroid [14], one of the most sophisticated dynamic taint tracking system, uses a modified Dalvik virtual machine to track flows of private data.

In this work, we investigate potential component leaks, the component-based information flows, which are different from the above approaches. Our results are generated by our previous work, PCLeaks, which performs information flow analysis through the known ICC vulnerabilities (e.g., Activity Hijacking). CHEX [27] and ContentScope [39] are two other tools that tackle potential component leaks, however CHEX limits itself to only considering leaks related to Activity hijacking while ContentScope only takes into account leaks related to Content Provider.

Machine learning based malware detection. Recently, Avdiienko et al. presented an approach [8] closely related to ours. Both their approach and ours take sensitive data flows as features for machine learning-based malware detection, and both rely on FlowDroid to extract sensitive data flows. However, instead of taking into account all intra-component leaks, we focus on component-based privacy leaks, the so-called potential component leaks. Besides, we take into account *SharedPreferences* in our study, which has not been considered in Avdiienko et al. approach. Furthermore, we have investigated the clustering impact of the training data set, which at the moment is rarely investigated in the literature. Allix et al. [2] empirically investigated the assessment of machine learning-based malware detectors for Android apps to measure the impact of datasets size and goodware/malware ratio, and the importance of validation scenarios. Our work is related in that we also measure the impact of several parameters and we raise one more factor to take into account when evaluating a malware detection approach: One specific approach may perform well only on a subset of Android applications.

Several other candidate features have been proposed to classify Android malware by using machine learning. For example, Peng et al. [34] apply probabilistic learning methods to the permissions of apps to detect malware. Gascon et al. [16] make use of embedded call graphs to build a malware detector. Other approaches [1], [6], [10], [15], [38] that rely on static or dynamic analysis also provide possible features for malware detection. Those features, along with the features we studied in this paper, could be combined to perform more accurate malware detection.

Empirical study on Android apps. In this work, we have empirically studied the distribution of potential component leaks. Empirical study provides a way of gaining knowledge quantitatively and qualitatively. Li et al. [24] presents an empirical study on how `Intent` is used in Android apps, showing that `Intent` is commonly used in Android apps. Ruiz et al. [31] show the prevalence of multiple advertisement libraries in Android apps. Liu et al. [26] studied on the safety of storing non-shared data on public storage of Android. Egele et al. [13] illustrate that 10,327 out of the 11,748 apps they studied contain at least one mistake in their usage of cryptographic APIs. Maji et al. [28] perform fuzz testing to evaluate the robustness of Android ICC mechanism, showing that exception handling is rarely used and that it is possible to crash an app at runtime from an unprivileged user process. Allix et al. [3] perform a forensic analysis of Android apps, showing evidences that many Android malicious apps are developed at an industrial scale.

VIII. Conclusion

In this study, we empirically investigated a new feature set for Android malware detection. This new feature set is based on potential component leaks (PCLs), which we define as sensitive data-flows that involve Android inter-component communications. We first showed that PCLs are common in Android apps. Then, further investigation showed that malicious apps contain significantly more PCLs than benign apps. Finally, we successfully applied PCLs as features for machine learning-based malware detection.

In this study, we also investigated the generalization of our PCLs-based feature set. Our results show that the performance of a malware detector is highly linked to the *quality* of its training set, and not only to the *quantity* aspects of the training set. In other words, providing more training data might well be a dead-end, since as shown by our results, some approaches seem to work well only for subsets of applications.

This study hence suggests a new direction to the community of machine learning-based malware detection. Instead of applying a feature set for all the apps in the wild, it could be better to only apply it for an appropriate set of apps (e.g., those apps that are somehow belonging to a same family). To that end, a new application could be first assigned to an appropriate cluster, and then be classified using a feature set specifically designed for that cluster.

Acknowledgments

This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under the project AndroMap C13/IS/5921289, by the BMBF within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, by the DFG's Priority Program 1496 Reliably Secure Software Systems and the project RUNSECURE.

References

[1] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *Security and Privacy in Communication Networks*, pages 86–103. Springer, 2013.

[2] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State, and Y. Le Traon. Empirical assessment of machine learning-based malware detectors for android. *Empirical Software Engineering*, pages 1–29, 2014.

[3] K. Allix, Q. Jérome, T. F. Bissyandé, J. Klein, R. State, and Y. Le Traon. A forensic analysis of android malware–how is malware written and how it could be detected? In *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*, pages 384–393. IEEE, 2014.

[4] Android Developers Documentation. Introduction to android. https://developer.android.com/guide/index.html. Accessed: 2015-03-18.

[5] AppBrain. Number of available android applications. http://www.appbrain.com/stats/number-of-android-apps, 2015. Accessed: 2015-02-23.

[6] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[7] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2014)*, 2014.

[8] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *International Conference on Software Engineering (ICSE)*, 2015.

[9] G. Canfora, F. Mercaldo, and C. A. Visaggio. A classifier of malicious android applications. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 607–614. IEEE, 2013.

[10] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. Mast: triage for market-scale mobile malware analysis. In *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, pages 13–24. ACM, 2013.

[11] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[12] D. Melanson – engadget.com. Eric schmidt: Google now at 1.5 million android activations per day. http://www.engadget.com/2013/04/16/liveblog-google-eric-schmidt-at-dive-into-mobile-2013/, 2013. Accessed: 2015-02-23.

[13] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[14] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, volume 10, pages 255–270, 2010.

[15] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *SIGSOFT FSE*, 2014.

[16] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.

[17] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *Proceedings of the 5th international conference on Trust and Trustworthy Computing*, TRUST'12, pages 291–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[18] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow analysis of android applications in droidsafe. 2015.

[19] J. Kim, Y. Yoon, K. Yi, and J. Shin. ScanDal: Static analyzer for detecting privacy leaks in android applications. In H. Chen, L. Koved, and D. S. Wallach, editors, *MoST 2012: Mobile Security Technologies 2012*, Los Alamitos, CA, USA, May 2012. IEEE.

[20] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6. ACM, 2014.

[21] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International Joint Conference on Artificial Intelligence*, 1995.

[22] L. Li, K. Allix, D. Li, A. Bartel, T. F. Bissyandé, and J. Klein. Potential Component Leaks in Android Apps: An Investigation into a new Feature Set for Malware Detection. In *The 2015 IEEE International Conference on Software Quality, Reliability & Security (QRS)*, 2015.

[23] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. Le Traon. ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In *The 30th IFIP International Conference on ICT Systems Security and Privacy Protection (SEC)*, 2015.

[24] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, 2015.

[25] L. Li, A. Bartel, J. Klein, and Y. Le Traon. Automatically exploiting potential component leaks in android applications. In *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2014)*. IEEE, 2014.

[26] X. Liu, Z. Zhou, W. Diao, Z. Li, and K. Zhang. An Empirical Study on Android for Saving Non-shared Data on Public Storage. In *The 30th IFIP International Conference on ICT Systems Security and Privacy Protection (SEC)*, 2015.

[27] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[28] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer. An empirical study of the robustness of inter-component communication in android. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

[29] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Statist.*, 18(1):50–60, 03 1947.

[30] McAfee Labs. Threats report. http://www.mcafee.com/in/resources/reports/rp-quarterly-threat-q3-2014.pdf, November 2014. Accessed: 2015-03-19.

[31] I. Mojica Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. Hassan. On the relationship between the number of ad libraries in an android app and its rating. *IEEE Software*, 2014.

[32] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[33] K. Pavneet Singh, T. F. D. A. Bissyande, D. Lo, and L. Jiang. An empirical study of adoption of software testing in open source projects. In *13th International Conference on Quality Software (QSIC)*, 2013.

[34] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 241–252. ACM, 2012.

[35] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[36] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. Management of an academic hpc cluster: The ul experience. In *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*, Bologna, Italy, July 2014. IEEE.

[37] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM conference on Computer and communications security (CCS 2014)*, 2014.

[38] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. AppContext: Differentiating Malicious and Benign Mobile App Behavior Under Contexts. In *International Conference on Software Engineering (ICSE)*, 2015.

[39] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium, NDSS2013*, 2013.